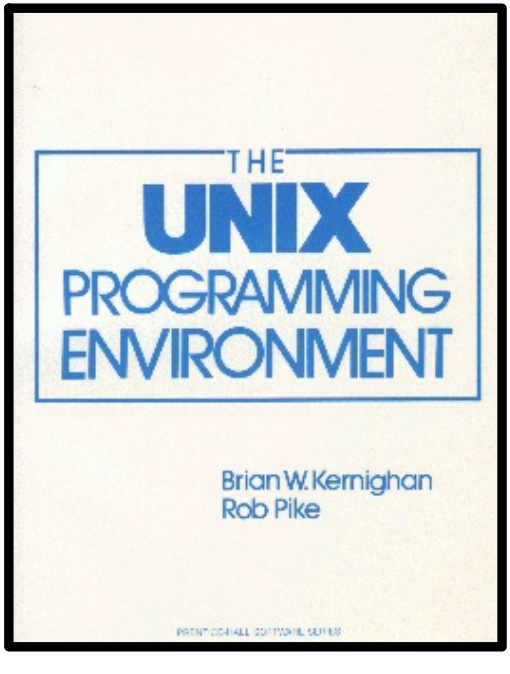
The UNIX Programming Environment



Remaster Ver.
11/25

Apuntes del libro The UNIX Programming Environment - 1984

versión digital-remaster de apuntes de hace 4 años :-)

0_ Featured commands

- ps → show running processes
- kill → kill processes
  - ↳ kill 0 : kill all shell session processes.
- nohup [command] → El commando se ejecuta a pesar de si se cierra el shell
- nice [command] → ejecuta [cmd] con low-cpu priority
- at [hora] [cmd] → ejecuta [cmd] a la hora [hora]
- od → octal dump
  - ↳ od -cb (file number + ASCII)
  - od --read-bytes=10 lee los primeros 10 bytes. El 2do byte MAY BE THE MAGIC NUMBER
- file [file] → devuelve el formato de [file]
- du → disk usage. Bytes que ocupa cada directorio
- grep cordinal /etc/passwd
  - ↳ Output: [Id. de Inicio]: llave codificada : UID: GID (group ID) : varios : dir de origen : Shell (/bin/sh)

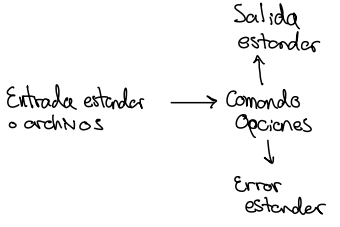
Env Variables

$PATH

↳ directorio donde el shell busca los programas

Add variable (zsh)

↳ export var_x="text"



Shell Operators

| → Pipe! Interconnects program outputs/inputs
↳ Si hay errores salen por stderr (default: terminal)

; → Sepcsa instrucciones

& → Ejecuta commando en background (y procede con los demás)
↳ ex: kwc book > out &
[1] 6184 ← pid of wc

wait ← espera que todos los commandos de & terminen.

Valores Ascii notables

\n : 012
\t : 011
\r : 015

TODOS SON ARCHIVOS

inode

La información de un archivo no está en el archivo en sí pero en su inode correspondiente. Los nombres de los archivos son links a inodes (inode 0 ← broken links). rm borra links, no archivos. Solo cuando un archivo ya no está linkeado, el kernel lo borra.

- ln [link-anterior] [nuevo-link]
  - ↳ crea links. Los links son indistinguibles. En cambio cp genera un nuevo inode.
- fechas inode → ultima lectura/ejecución.
  - → ultima alteración inode.
  - → ultima modificación.
- touch file → actualiza fecha modificación file.

ls
- -lc → fecha del inode
- -lu → fecha ultimo acceso
- -l → modo lista
- -t → ordena por fecha.
- -i → muestra inode de cada archivo

More Commands

- who, date, wc
  - who → running users
  - date → current date
  - wc → word count
- grep, sort, tail
  - grep → search content
  - sort → sort content
  - tail → ↳ show end of file; tail -1 file
- cmp f1 f2 → compare two files (boolean function)
- diff f1 f2 → verbose cmp
- df → disk free
- dd : conversión y copia de archivo
- echo -n → omite \n.

Permisos

- d/- (---)(---)(---) {r, w, x}
  - read, write, execute
- ls -d → directory info.
  - ↳ en directorios 'x' es que se pueden buscar archivos allí
- chmod permissions ___ [file]
  - 4: lectura
  - 2: escritura
  - 1: ejecución
  - chmod +x -w
- solo el owner puede cambiar los permisos de un archivo

Shell

ejecuta en el mismo shell

- * → match any string
- a[1234] → a1 a2 a3 a4
- a[1-4] → ↑
- > → coloca stdin sobre un archivo
- >> → mismo pero modo append.
- < → file as input

(cmd) vs {cmd}
- (cmd) → ejecuta en subshell

Subshells

zsh [file] // equivalente: zsh < file

> crea una instancia de shell dentro del shell donde se ejecutan los comandos del archivo

Parametros shell

script hello there friend ... $9 (max)
$1 $2 $3

- Use " " to avoid space char problems : _
- Todos los args: $#
- /dev/null → devuelve 0. Sirve para mandar cosas a la basura
- tee [file] ← recibe input, la escribe en file y la escupe.
  - ↳ echo "hola mundo" | tee hola.txt | wc
- Pipe | tiene mas precedencia que ;. Usar parentesis si necesario
  - ↳ (date; who) | tee save | wc
  - va a imprimirse antes
- && permite ejecución en paralelo: (sleep 30; date) && who
- && → ejecución a condición
  - ↳ p1 && p2 #ejecuta p2 si p1 es true
  - ↳ opuesto: p1 || p2 #si p1 falla (false)
  - ↳ comentario
- ? ← match cualquier caracter individual
- read var
  - ↳ Asigna var por input
  - ej: echo Hi | read var; now Hi.

Redireccionamiento de I/O

Todo programa tiene 3 archivos estandar creados al momento de ejecución y numerados 0, 1 y 2. Se llaman descriptores de archivos

- 0 → stdin
- 1 → stdout
- 2 → stderr

Ex: cmd 1>results 2>error_logs 2>&1
- 2>&1 (por defecto 2>&1)
  - ↳ errores a stdout!
  - desplegar errores en flujo de stdout
  - colocar stderr en el mismo flujo que stdout

- Ejemplos
  - n>&m : mezclar la salida del descriptor n con la del desc. m.
  - n<&m : mezclar la entrada del desc. n con el desc. m

Command Output

- Se puede llamar usando backquotes
  - ↳ ex: echo `date` => 27/8/1982
    Interpreta dentro y usa salida

Shell Variables

- Los variables de los argumentos son read-only
  Var1=`echo test`
- Para pasar argumentos del shell padre al shell hijo usamos export var1
- El shell hijo NO puede MODIFICAR los variables del shell padre
- cmd shift corre parametros 1 lugar.

Filters

- Programe that read input, processit and outputs it.
  - ↳ grep, wc, tail, sort, etc.
- Sort
  - -f : A=a
  - -d: Ignora todo caracter menos letras
  - -n: clasifica numericamente
  - -r: Invierte sentido
  - +m: excluye primeros m lineas.
  - -u: suprime lineas identicas
- Uniq (descarta valores duplicados)
  - -d (duplicate) Imprime solo renglones repetidos
  - -u (Imprime los que no se repiten)
  - -c cuenta numero de ocurrencias de cada linea.
- comm f1 f2 (compara archivo f1 con f2)
- tr (translitera entrada)
  - ↳ ex: tr a-z A-Z <f1 >f2
    (cambia f1 a mayusculas, guarda en f2)

Awk

(lenguaje de filtro inspirado a C) ("se asemeja a C pero se puede costo)

[No incluido - obsoleto]

La variable $0 es la linea entera

ex: awk '/regex/ {print}' file1

{s=s+$1} END {print s}
↳ suma de valores de una columna

$n
field n (es por columnas!)

Sed (Stream editor)

Lee columna por columna, aplica patron y devuelve modificado

[obsoleto - No incluido]

Ex: sed s/old/new/g #reemplaza old por new
sed -n '20,30p' # Imprime lineas de 20 a 30
sed `1,10d` # borra 10 primeros lineas

Grep

Operador clausura *
- ↳ ex: X* → cadena de Xxxx...
  * → cualquiera
  *X → cualquiera que termine en X.

Las expresiones grep se evaluan linea - por - linea

grep ↗ fgrep → busca varias strings en simultaneo
egrep → para regex

-f → permite cargar un archivo con patrones
↳ cat file | fgrep -f archivo_de_patrones

For

- Unico iterador que no requiere archivo

Syntax:
```
for var in lista_de_palabras do
    comandos
done.
```

- one liner : for i in lista; do [comando]; done

Regex

^ : match start Usar " " para evitar mexclar al shell.

$: match end

[^0-9]*: cualquier tira sin números.
not

Operadores: OR → this | that
+ → una o más aparaciones
? → ninguna o una aparición

Cool examples

^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o...*$
↳ Palabras con vocales en orden

(r) : anidación de expresiones

Shell env.vars

todos los args en una lista

1. $# → number of args
2. $* → todos los args en un string / $@
3. $PS1 → cadena de simbolo de espera ($)
4. $PS2 → cadena de símbolo de continuación (>)
5. $HOME → default de cd
6. $ PATH → lista dirs para comandos.

- comando set #sin args muestra el valor de todas las variables del ambiente

IFS: Internal field separater
↳ Normalmente el espacio " ".

set `date`
↳ $1: día 3$año
$2: mes

Evaluate Shell Vars

- ${var} → permite evaluar variables Note: $varx ≠ ${var}x
- ${var-thing} → valor de var si existe, else, thing.
- ${var=thing} → valor de var si existe, else, thing, and var ← thing.
- ${var?msg} → valor de var si existe, else se despliega msg y se cierra shell
- ${var+thing} → thing si var existe, else nada.
- If conditionals for scripts.

Signals

Trap: Detección de interrupciones
↳ ex: trap 'echo hi' 15; prog
↳ will command signal

#ejecuta sth al recibir señal.

0: all ok	3: abandonar (ctrl^\)	UNIX
1: corte	9: cancelar	ctrl^s: stop
2: interrupt	15: terminar (by will)	ctrl^q: resume

- kill -9 PID : send signal 9. Process will quit UNGRACEFULLY.

Conditions Shell

```
for i in [...]
do
   (...)
done
```

```
while cmd   (true)
do
   (...)
done
```

```
until cmd   (false)
do
   (...)
done
```

Obs: sleep devuelve true al completarse

":" → evalua args y devuelve true, clave para ciclo infinito

```
while :
do
   (...)
done
```

* → cualquier cosa
? → cualquier caracter
a|b → caracter a OR b.

for (exp1; cond1; exp2)

<ctype.h>

- Define pruebas a tipos de datos

ex: isalpha(c) → si es digito
isupper(c) → si es MAYUS
islower(c) → si es MINUS
isdigit(c) → si es NUMERIC
isprint(c) (char imprimible)

Programación con E/S standard (C)

- rutinas básicas con I/O
  - getchar() → extrae el sig caracter de stdin.
  - putchar(c) → pone caracter c en la salida estander.
- printf (print formatter)
  printf("%o", var)
  ↳ %o, %d, %f, ...
  octal, decimal, float
  %6.3f → width 6, precision 3
  %030 → assigned space, replace with 0s.
- Parte de <stdio.h>
- int argc, char *argv[]
  #args stdin
  list of args

ejemplo:

```
{
  int c;
  while ((c=getchar()) != EOF)
    if (isascii(c) && (isprint(c)||c=='\n'))
      putchar(c);
    else
      printf("\\%030",c);
}
return 0;
```

special char. end of file

- For working with strings
  - strcat(s,t) → concatena t a s, returns s.
  - strncat(s,t) → concatena n caracteres de t a s.
  - strcpy(s,t) → copia t en s.
  - strcmp(s,t) → compara t con s.
  - strlen(s) → length of s
  - strchr(s,c) → regresa char * del primer caracter de C en s
  - strrchr(s,c) → regresa char * del primer char de C en s
- Memory
  - calloc(n,m): regresa un puntero de num bytes
  - free(p): libera memoria asignada a puntero p
- Variables standard
  - EOF # end of file, usualmente -1
  - NULL # invalid pointer, usually 0
  - BUFSIZ # tamaño normal del buffer
  - feof(fp) → non-zero si hay EOF en flujo fp
  - ferror(fp) → non-zero si hay Error en flujo fp
- Funciones utiles (UNIX)
  - mktmp("name.xxx"); → fopen(tmp,"w")
    ↳ crea archivo temporal. Unicidad garantizada
  - sprintf(buf "%s,...", str1) → guarda salida formateada
  - value = getenv("HOME") //almacena valor de $HOME

Para compilar: cc -o prog prog.c
↳ reemplaza a "a.out" assembled.out

Ex: while (cond) { }
  if (c) { } else { }
  switch (var) {
    case n: (...)
  }

Files in C

- un puntero a un archivo: `FILE* fp;`
  para abrirlo: `fp = fopen(name, mode);` → {r, w, a}
  Si: file doesn't exist, fp = NULL;
- `c = getc(fp)` → toma el sig. caracter del flujo fp y lo asigna a c.
- `putc(c, fp)` → coloca c como el sig. char que apunta fp.
  Obs: `getchar() ≈ getc(stdin)`
  `putchar(c) ≈ putc(c, stdout)`
- archivos: open, creat, unlink, close (unlink → borra archivo)

Processes

```
#import <unistd.h>
char* getlogin();   → devuelve nombre de quien controla shell del proceso.
```

Para crear procesos en bajo nivel: execlp y execvp

- Ejecuta otro prog y no regresa
- Solo regresa si hay error.
- útil si desconoce argumentos.

`execlp("date", "date", ..., (char*)0);`
`execvp("filename", args)`

- nombre del archivo
- prog. name
- argumentos (terminar en 0)
- literal argv. argv[0] prog. name

ex: `execlp("/bin/sh", "sh", "-c", cmd, (char*)0);`
→ trata cmd como una línea completa (no separa)

- Control de procesos: fork y wait
  ↳ permite volver a parent tras execlp.
  - `proc_id = fork()` // Separa el programa en 2 copias que continuan ejecutandose. Se diferencian por el process_id
  - Si hay error fork retorna -1.
  - Ex:
    `if (fork()==0) execlp(...);` ↳ El hijo es 0, el padre ≠0.
  - `wait(&status); #espera fin de fork.`
    ↳ 0 para terminación ok.
    No maneja errores, anormalidades o multifork.
  - Recordar cuando se fork procesos que el padre ignore señales de interrupción;

```
if (fork()==0)
    execlp(...);
signal(SIGINT, SIG_IGN);
wait(&status);
signal(SIGINT, SIG_DFL);
```

Yacc

[reseña, legacy]

- Yet another compiler compiler es un generador de parsers. Toma un set de reglas de lenguaje y te da un parser.
- `yacc -o name.c rules.y`
  ↳luego compilo parser con cc.

Steps:
1. Definir gramática
2. Para cada regla, acción (en C)
3. Analizador lexicográfico que tokeniza la entrada (`yyparse()`, `yylex()`)
4. Rutina de control para llamar al analizador.

Parsers complejos, muchos archivos... ← Para cosas más complejas existe Lex.

UNIX Syscalls

- Low level I/O: llamadas directas al kernel.
  Este mete datos en buffers optimizados según dispositivo
- File descriptor: `filero(fp)` # returns file descriptor del archivo (fp: FILE*)
  0 → In
  1 → Std out
  2 → err
  (Por defecto enlazados con /dev/tty)
- I/O esencial → `read(fd, buf, n)` / `write(fd, buf, n)`
  Ex: Mostrar datos que serán escribiendo

```
char buf[size];
int n;
for (;;) {
    while((n=read(0, buf, sizeof buf))>0)
        write(1, buf, n);
    sleep(10);
}
```

(0: stdin; 1: stdout; sleep → cool CPU)

- Obs: varios procesos pueden editar/ leer #bytes un archivo en simultaneo. Si un read devuelve 0, pero otro prog escribe, el siguiente read no será vacío.
- Obs: cuando un proceso tiene hijos, los file desc. del padre 3 Se mantienen abiertos y hay un solo buffer de I/O.
  Por lo que si dos procesos leen el archivo, suponiendo sincronía, cada uno leerá los bytes pares/impares resp.
- Set jump: permite hacer saltos fuera de funciones. (super costo)

```
#include <setjmp.h>
jmp_buf jumpeg; // almacena p/a
setjmp(jumpeg); // guarda p/a
longjmp(jumpeg, 0); // salta p/a guardada.
```

UNIX Signals

Las señales más comunes son: Interrupt, Termination, Hangup, Kill

- Muchas señales generan un coredump para revisión posterior.
- La syscall `signal(sig_num, func)` altera el comportamiento tras una señal, ejecutando func al lanzarse señal.
  Definiciones en <signal.h>
- Util para cerrar procesos huerfanos antes de terminar el prog.
- Ex: Ignorar Interrupts: `signal(SIGINT, SIG_IGN);`
  Default Interrupt signal: `signal(SIGINT, SIG_DFL);`
- Alarmas: permite generar un timeout
  ex: `alarm(sec);`
  `signal(SIGALARM, onalarm);`
- Verificar que SIGINT no se ignora:

```
if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, g);
```